Most testers I know have heard of mutation testing, but few have actually performed it. Mutation testing has the reputation of being difficult and requiring expensive third-party software tools. However, in this month’s column, I’ll show you how to create a super-simple (fewer than two pages of code and four hours of time) mutation testing system using C# and Visual Studio. By keeping the mutation testing system simple, you can get most of the benefits of a full-fledged mutation system at a fraction of the time and effort.

Mutation testing is a way to measure the effectiveness of a set of test cases. The idea is simple. Suppose you start with 100 test cases and your system under test (SUT) passes all 100 tests. If you mutate the SUT—for example, by changing a “>” to a “<” or by changing a “+” to a “-”—you presumably introduce a bug into the SUT. Now if you rerun your 100 test cases, you’d expect at least one test case failure indicating that one of your test cases caught the faulty code. But if you see no test failures, then it’s quite likely that your set of test cases missed the faulty code and didn’t thoroughly exercise the SUT.

The best way for you to see where I’m headed is by looking at **Figure 1**.

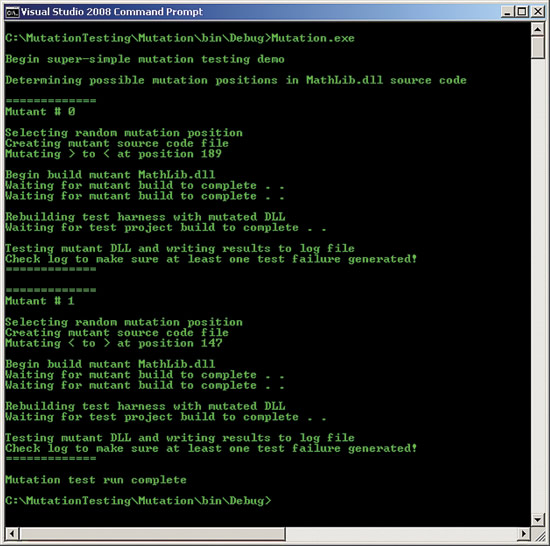


Figure 1 **Mutation Testing Demo Run**

The SUT in this example is a library named MathLib.dll. The technique I present here can be used to test most Microsoft .NET Framework systems including DLLs, WinForms applications, ASP.NET Web applications and so on. The mutation system begins by scanning the original source code for the SUT, looking for candidate code to mutate. My super-simple system looks only for “<” and “>” operators. The test system is set to create and evaluate two mutants. In a production scenario, you’d likely create hundreds or even thousands of mutants. The first mutant randomly selects an operator to mutate, in this case a “>” operator at character position 189 in the SUT source code, and mutates the token to “<”. Next, the mutant DLL source code is built to create a mutant MathLb.dll library. Then the mutation system calls a suite of test cases on the mutant SUT, logging results to a file. The second iteration creates and tests a second mutant in the same way. The result of the log file is:

**=============  
Number failures = 0  
Number test case failures = 0 indicates possible weak test suite!  
=============  
Number failures = 3  
This is good.  
=============**

The first mutant didn’t generate any test case failures, which means you should examine the source code at position 189 and determine why none of your test cases exercise that code.

## The SUT

My super-simple mutation testing demo consists of three Visual Studio projects. The first project holds the SUT, and in this case is a C# class library named MathLib. The second project is a test harness executable, in this case a C# console application named TestMutation. The third project creates and builds the mutants, in this case a C# console application named Mutation. For convenience I placed all three projects in a single directory named MutationTesting. With mutation testing there are a lot of files and folders to keep track of and you shouldn’t underestimate the challenge of keeping them organized. For this demo I used Visual Studio 2008 (but any Visual Studio version will work) to create a dummy MathLib class library. The entire source code for the dummy SUT is shown in **Figure 2**.

Figure 2 **The Entire Dummy SUT Source Code**

1. **using** System;
2. **namespace** MathLib
3. {
4. **public** **class** Class1
5. {
6. **public** **static** **double** TriMin(**double** x, **double** y, **double** z)
7. {
8. **if** (x < y)
9. **return** x;
10. **else** **if** (z > y)
11. **return** y;
12. **else**
13. **return** z;
14. }
15. }
16. }

Notice I retained the default class name of Class1. The class contains a single static method, TriMin, which returns the smallest of three type double parameters. Also note the SUT is deliberately incorrect. For example, if x = 2.0, y = 3.0 and z = 1.0, the TriMin method returns 2.0 instead of the correct 1.0 value. However, it’s important to note that mutation testing does notdirectly measure the correctness of the SUT; mutation testing measures the effectiveness of a set of test cases. After building the SUT, the next step is to save a baseline copy of the source file, Class1.cs, to the root directory of the mutation testing system. The idea is that each mutant is a single modification of the original source code of the SUT and so a copy of the original SUT source must be maintained. In this example I saved the original source as Class1-Original.cs in directory C:\MutationTesting\Mutation.

## The Test Harness

In some testing situations, you may have an existing set of test-case data, and in some situations you have an existing test harness. For this super-simple mutation testing system, I created a C# console application test harness named TestMutation. After creating the project in Visual Studio, I added a Reference to the SUT: MathLib.dll located at C:\MutationTesting\MathLib\bin\Debug. The entire source code for the test harness project is presented in **Figure 3**.

Figure 3 **The Test Harness and Test Data**

1. **using** System;
2. **using** System.IO;
4. **namespace** TestMutation
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. **string**[] testCaseData = **new** **string**[]
11. { "1.0, 2.0, 3.0, 1.0",
12. "4.0, 5.0, 6.0, 4.0",
13. "7.0, 8.0, 9.0, 7.0"};
15. **int** numFail = 0;
17. **for** (**int** i = 0; i < testCaseData.Length; ++i) {
18. **string**[] tokens = testCaseData[i].Split(',');
19. **double** x = **double**.Parse(tokens[0]);
20. **double** y = **double**.Parse(tokens[1]);
21. **double** z = **double**.Parse(tokens[2]);
22. **double** expected = **double**.Parse(tokens[3]);
24. **double** actual = MathLib.Class1.TriMin(x, y, z);
25. **if** (actual != expected) ++numFail;
26. }
28. FileStream ofs = **new** FileStream("..\\..\\logFile.txt",
29. FileMode.Append);
30. StreamWriter sw = **new** StreamWriter(ofs);
31. sw.WriteLine("=============");
32. sw.WriteLine("Number failures = " + numFail);
33. **if** (numFail == 0)
34. sw.WriteLine(
35. "Number test case failures = " +
36. "0 indicates possible weak test suite!");
37. **else** **if** (numFail > 0)
38. sw.WriteLine("This is good.");
39. sw.Close(); ofs.Close();
40. }
41. }
42. }

Observe that the test harness has three hardcoded test cases. In a production environment, you’d likely have many hundreds of test cases stored in a text file and you could pass the filename in to Main as args[0]. The first test case, “1.0, 2.0, 3.0, 1.0,” represents the x, y and z parameters (1.0, 2.0 and 3.0), followed by the expected result (1.0) for the TriMin method of the SUT. It’s obvious the test set is inadequate: Each of the three test cases is basically equivalent and has the smallest value as the x parameter. But if you examine the original SUT, you’ll see that all three test cases would in fact pass. Will our mutation testing system detect the weakness of the test set?

The test harness iterates through each test case, parses out the input parameters and the expected return value, calls the SUT with the input parameters, fetches the actual return value, compares the actual return with the expected return to determine a test case pass/fail result, and then accumulates the total number of test case failures. Recall that in mutation testing, we’re primarily interested in whether there’s at least one new failure, rather than how many test cases pass. The test harness writes the log file to the root folder of the calling program.

## The Mutation Testing System

In this section, I’ll walk you through the mutation testing program one line at a time, but omit most of the WriteLine statements used to produce the output shown in **Figure 1**. I created a C# console application named Mutation in the root MutationTesting directory. The program begins with:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.IO;
4. **using** System.Diagnostics;
5. **using** System.Threading;
7. **namespace** Mutation
8. {
9. **class** Program
10. {
11. **static** Random ran = **new** Random(2);
12. **static** **void** Main(**string**[] args)
13. {
14. **try**
15. {
16. Console.WriteLine("\nBegin super-simple mutation testing demo\n");
17. ...

The purpose of the Random object is to generate a random mutation position. I used a seed value of 2, but any value will work fine. Next, I set up the file locations:

string originalSourceFile = "..\\..\\Class1-Original.cs";

string mutatedSourceFile = "..\\..\\..\\MathLib\\Class1.cs";

string mutantProject = "..\\..\\..\\MathLib\\MathLib.csproj";

string testProject = "..\\..\\..\\TestMutation\\TestMutation.csproj";

string testExecutable =

"..\\..\\..\\TestMutation\\bin\\Debug\\TestMutation.exe";

string devenv =

"C:\\Program Files (x86)\\Microsoft Visual Studio 9.0\\Common7\\IDE\\

devenv.exe";

...

You’ll see how each of these files is used shortly. Notice that I point to the devenv.exe program associated with Visual Studio 2008. Instead of hardcoding this location, I could have made a copy of devenv.exe and placed it inside the mutation system root folder.

The program continues:

1. List<**int**> positions = GetMutationPositions(originalSourceFile);
2. **int** numberMutants = 2;
3. ...

I call a helper GetMutationPositions method to scan through the original source code file and store the character positions of all “<” and “>” characters into a List, and set the number of mutants to create and test to two.

The main processing loop is:

1. **for** (**int** i = 0; i < numberMutants; ++i) {
2. Console.WriteLine("Mutant # " + i);
3. **int** randomPosition = positions[ran.Next(0, positions.Count)];
4. CreateMutantSource(originalSourceFile, randomPosition, mutatedSourceFile);
6. **try** {
7. BuildMutant(mutantProject, devenv);
8. BuildTestProject(testProject, devenv);
9. TestMutant(testExecutable);
10. }
11. **catch** {
12. Console.WriteLine("Invalid mutant. Aborting.");
13. **continue**;
14. }
15. }
16. ...

Inside the loop, the program fetches a random position of a character to mutate from the List of possible positions and then calls helper methods to generate mutant Class1.cs source code, build the corresponding mutant MathLib.dll, rebuild the test harness so that it uses the new mutant and then test the mutant DLL, hoping to generate an error. Because it’s quite possible that mutated source code may not be valid, I wrap the attempt to build and test in a try-catch statement so I can abort the testing of non-buildable code.

The Main method wraps up as:

1. ...
2. Console.WriteLine("\nMutation test run complete");
3. }
4. **catch** (Exception ex) {
5. Console.WriteLine("Fatal: " + ex.Message);
6. }
7. } // Main()

## Creating Mutant Source Code

The helper method to get a list of possible mutation positions is:

1. **static** List<**int**> GetMutationPositions(**string** originalSourceFile)
2. {
3. StreamReader sr = File.OpenText(originalSourceFile);
4. **int** ch = 0; **int** pos = 0;
5. List<**int**> list = **new** List<**int**>();
6. **while** ((ch = sr.Read()) != -1) {
7. **if** ((**char**)ch == '>' || (**char**)ch == '<')
8. list.Add(pos);
9. ++pos;
10. }
11. sr.Close();
12. **return** list;
13. }

The method marches through the source code one character at a time looking for greater-than and less-than operators and adding the character position to a List collection. Notice that a limitation of this super-simple mutation system as presented is that it can only mutate single-character tokens such as “>” or “+” and can’t deal with multicharacter tokens such as “>=”. The helper method to actually mutate the SUT source code is listed in **Figure 4**.

Figure 4 **The CreateMutantSource Method**

1. **static** **void** CreateMutantSource(**string** originalSourceFile,
2. **int** mutatePosition, **string** mutatedSourceFile)
3. {
4. FileStream ifs = **new** FileStream(originalSourceFile, FileMode.Open);
5. StreamReader sr = **new** StreamReader(ifs);
6. FileStream ofs = **new** FileStream(mutatedSourceFile, FileMode.Create);
7. StreamWriter sw = **new** StreamWriter(ofs);
8. **int** currPos = 0;
9. **int** currChar;
11. **while** ((currChar = sr.Read()) != -1)
12. {
13. **if** (currPos == mutatePosition)
14. {
15. **if** ((**char**)currChar == '<') {
16. sw.Write('>');
17. }
18. **else** **if** ((**char**)currChar == '>') {
19. sw.Write('<');
20. }
21. **else** sw.Write((**char**)currChar);
22. }
23. **else**
24. sw.Write((**char**)currChar);
26. ++currPos;
27. }
29. sw.Close(); ofs.Close();
30. sr.Close(); ifs.Close();
31. }

The CreateMutantSource method accepts the original source code file, which was saved away earlier, along with a character position to mutate and the name and location of the resulting mutant file to save to. Here I just check for “<” and “>” characters, but you may want to consider other mutations. In general, you want mutations that will produce valid source, so, for example, you wouldn’t change “>” to “=”. Also, mutating in more than one location isn’t a good idea, because just one of the mutations might generate a new test case failure, suggesting that the test set is good when in fact it might not be. Some mutations will have no practical effect (such as mutating a character inside a comment), and some mutations will produce invalid code (such as changing the “>>” shift operator to “><”).

## Building and Testing the Mutant

The BuildMutant helper method is:

1. **static** **void** BuildMutant(**string** mutantSolution, **string** devenv)
2. {
3. ProcessStartInfo psi =
4. **new** ProcessStartInfo(devenv, mutantSolution + " /rebuild");
5. Process p = **new** Process();
7. p.StartInfo = psi; p.Start();
8. **while** (p.HasExited == **false**) {
9. System.Threading.Thread.Sleep(400);
10. Console.WriteLine("Waiting for mutant build to complete . . ");
11. }
12. p.Close();
13. }

I use a Process object to invoke the devenv.exe program to rebuild the Visual Studio solution that houses the Class1.cs mutated source code and produces the MathLib.dll mutant. Without arguments, devenv.exe launches the Visual Studio IDE, but when passed arguments, devenv can be used to rebuild Projects or Solutions. Notice I use a delay loop, pausing every 400 milliseconds, to give devenv.exe time to finish building the mutant DLL; otherwise the mutation system could attempt to test the mutant SUT before it’s created.

The helper method to rebuild the test harness is:

1. **static** **void** BuildTestProject(**string** testProject, **string** devenv)
2. {
3. ProcessStartInfo psi =
4. **new** ProcessStartInfo(devenv, testProject + " /rebuild");
5. Process p = **new** Process();
7. p.StartInfo = psi; p.Start();
8. **while** (p.HasExited == **false**) {
9. System.Threading.Thread.Sleep(500);
10. Console.WriteLine("Waiting for test project build to complete . . ");
11. }
12. p.Close();
13. }

The main idea here is that, by rebuilding the test project, the new mutant SUT will be used when the test harness executes rather than the previously used mutant SUT. If your mutant source code is invalid, BuildTestProject will throw an Exception.

The last part of the super-simple mutation testing system is the helper method to invoke the test harness:

1. ...
2. **static** **void** TestMutant(**string** testExecutable)
3. {
4. ProcessStartInfo psi = **new** ProcessStartInfo(testExecutable);
5. Process p = **new** Process(); p.StartInfo = psi;
6. p.Start();
7. **while** (p.HasExited == **false**)
8. System.Threading.Thread.Sleep(200);
10. p.Close();
11. }
13. } // class Program
14. } // ns Mutation

As I mentioned earlier, the test harness uses a hardcoded log file name and location; you could parameterize that by passing information as a parameter to TestMutant and placing it inside the Process start info, where it would be accepted by the TestMutation.exe test harness.

## A Real-World, Working Mutation Testing System

Mutation testing is simple in principle, but the details of creating a full-fledged mutation testing system are challenging. However, by keeping the mutation system as simple as possible and leveraging Visual Studio and devenv.exe, you can create a surprisingly effective mutation testing system for .NET SUTs. Using the example I’ve presented here, you should be able to create a mutation testing system for your own SUTs. The primary limitation of the sample mutation testing system is that, because the system is based on single-character changes, you can’t easily perform mutations of multicharacter operators, such as changing “>=” to its “<” complement operator. Another limitation is that the system only gives you the character position of the mutation, so it doesn’t provide you with an easy way to diagnose a mutant. In spite of these limitations, my sample system has been used successfully to measure the effectiveness of test suites for several midsize software systems.